Multifidelity Memory System Simulation in SST

Patrick Lavin
Sandia National Laboratories
Albuquerque, NM, USA
prlavin@sandia.gov

Jeffrey Young
Georgia Institute of Technology
School of Computer Science
Atlanta, GA, USA
jyoung9@gatech.edu

Richard Vuduc
Georgia Institute of Technology
School of Computational Science and Engineering
Atlanta, GA, USA
richie@cc.gatech.edu

ABSTRACT

As computer systems grow larger and more complex, it takes more time to simulate their behavior in detail. Researchers interested in simulating large-scale systems must choose between less-accurate high-level models or simulating smaller portions of their benchmark suite, both of which are highly manual, offline approaches that require time-consuming analysis by experts. Multifidelity simulation aims to lessen this burden by automatically adapting the fidelity of a simulation to the complexity of the behavior occurring at any given point in time. We show how a multifidelity memory system model can be used to accelerate single node simulation by up to 2x with 1-5% mean absolute percent error in the simulated instructions per cycle across benchmark suites.

CCS CONCEPTS

• **Computing methodologies** → **Discrete-event simulation**; • **Computer systems organization** → **Architectures**.

KEYWORDS

Architectural simulation, statistical simulation, memory modeling, phase detection

1 INTRODUCTION

We consider the problem of how to speed up cycle-level simulations, which attempt to capture the cycle-by-cycle behavior of the components deemed most critical to the performance of the system. We propose a technique, which we refer to as *multifidelity simulation*, that trains and utilizes low-fidelity statistical models during periods of easy-to-predict behavior. For memory-system simulations, we show that it is possible to speed up end-to-end simulations by up to 2x with 1-5% mean absolute percent error in the simulated instructions per cycle across benchmark suites.

Our focus is on cycle-level simulation as it lies in the area between high-level analytical models [12, 27], which can be hard to adapt to new systems and can miss the dynamic behavior that is difficult to capture, and much more detailed RTL-level or gate-level simulation which is far too expensive to use early in the design cycle. Cycle-level simulation allows us to run simulations in days or hours, but still includes a high amount of detail about the system, making it the ideal place to work towards creating a co-design workflow, where system designs and algorithms are changed in tandem.

As processors grow more complex, detailed simulations too take longer, as each processor has more components that need to be considered, and system designers want to simulate more processors working in parallel. While we wish to maintain the flexibility of a detailed simulation, we can at the same time realize that we don't need the full detail of such a simulation for the entirety of a program. Fortunately, there exist methods of simulating less than the entire program in full detail, which we will cover in Section 2. However, our approach is distinct from most existing work, in that it aims to automatically adapt the level of detail of the simulation to the complexity of the behavior of the simulated system. In physical simulations, such techniques are known as multifidelity simulations as they utilize multiple models of of the same behavior. For instance, you may imagine a traffic simulation that has a *low-fidelity* model that represents traffic as a flow, which is enough to estimate average statistics when traffic is moving smoothly, as well as a *high-fidelity* model that includes detailed descriptions of driver personality [21] for when roads are congested. Our goal in this work is to adapt the ideas of multifidelity simulation to memory system modeling.

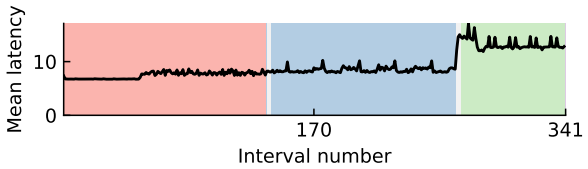

Figure 1: An example memory access latency trace of the `deriche` benchmark from PolyBench. Colors denote phases identified by the online phase detector.

To achieve a multifidelity memory system simulation, we will adapt ideas from recent work on multifidelity cache models [14]. Our contributions include (1) the use of statistical techniques to identify regions of programs that are suitable for use in training low-fidelity models, (2) the implementation of multifidelity components such as phase detection and stability detection in a widely-used simulator, SST [25], and (3) a multifidelity memory system model. Beyond the aforementioned 2x speedup and 1-5% average error, we demonstrate the model's ability to automatically capture input-dependent behavior.

2 RELATED WORK

Due to the popularity of simulation in the process of computer system design, numerous attempts have been made to accelerate it. Some techniques are focused on simulating a single node, as our approach is, but there are even lower-fidelity models used in the context of large-scale simulation that are related to our work as well, as we see our work as a way to bridge the gap between detailed single-node simulation, and the higher-level simulations used for modeling entire supercomputers.

2.1 Single node simulation

A single node, cycle-level simulation models a processor together with its memory system and any accelerators. We can categorize techniques related to our own into two categories: those that reduce the number of instructions simulated, and those that reduce the fidelity of the simulation.

2.1.1 Reducing instruction count. To reduce the number of instructions simulated, there are popular sampling techniques: *statistical sampling* and *clustering-based sampled simulation*. Statistical sampling reduces the number of instructions simulated by choosing intervals of the program to at random to simulate in full-detail, fast-forwarding between them using functional simulation which advances the program without modeling what is happening in the simulated components[1]. This technique has the benefit of providing rigorous error bars on the predicted performance of the program. Statistical sampling was pioneered by Conte et al. [6], and the most modern implementation is QFlex, which uses parallel simulation and FPGAs to further accelerate simulation [22]. While we do not provide the error bars that they do, our approach adapts online to the complexity of the behavior of the program. It is likely that our approach could be combined with statistical sampling to get the benefits of both.

Related to this technique is clustering-based sampled simulation, which first simulates the entire program in detail, and then performs an offline analysis to cluster related regions of the program. Then the program can be simulated much more efficiently by only simulating those representative samples. The seminal work in this area is SimPoint [24]. Barrierpoint[4] extends this method to multi-threaded, barrier-based programs, and LoopPoint further extends this arbitrary parallel constructs. However, these approaches are offline, meaning that simulating the entire program after making changes to it or changing the input will always be required, in contrast to our work.

2.1.2 Reducing fidelity. Another approach to increasing the speed of simulation is through the use of higher-level models which don't consider as much detail about the operations of the core (what we would call a lower-fidelity model). The best example of such as simulator is Sniper [3], which uses an "interval" core model. In their model, they make the observation that it is very easy to predict core performance when there are no miss events, such as TLB and L1 cache misses. As such, they do not simulate each instruction in detail as it moves through an out-of-order processor, rather they model when these miss events will happen. They report an order of magnitude improvement in runtime over cycle-accurate in this technique. However, such techniques make many assumptions about the design of the memory hierarchy and are therefore require significant effort to alter if one wants to simulate a radically different memory hierarchy.

To our knowledge, there are two existing multifidelity computer architecture simulations: ^{2}Sim [16], and one by Lavin et al. [14]. In ^{2}Sim, when a basic block is first encountered, it is run in a detailed out-of-order simulator and information is collected about the block. They record the number of cycles spent in the block both in the core and out of the core. On future invocations, the out-of-order core model does not need to be run and only the cache is simulated to get the off-core timings. The framework presented by Lavin et al. is similar in it also first runs models in full-detail before switching to lower-fidelity models. However, their framework is more general and can be applied to more components of the simulation than just the core, which is why it is the one we will be extending in this work to include the entire memory system.

[1]Simulators such as Gem5 [2] support this fast-fowarding by supporting both detailed and functional core models and the ability to swap between them during simulation, but users must decide for themselves when this switching happens.

2.2 Large-scale simulation

In the world of large-scale simulation, such as simulations of 100's or 1000's of nodes, even less fidelity is used due to the time it would take to simulate nodes in detail. There are two approaches to modeling the node behavior that we are aware of: SST's Ember library [11] and SST/Macro [1]. While these approaches do not include much detail on the node model, they are indeed low-fidelity models. In both Ember simulations and the skeletonized simulations of SST/Macro, the network is simulated in detail, but the compute is simulated with a simple delay. Calculating an appropriate delay to represent the computation is a manual process, requiring architects to perform studies determining how each compute region behaves as a function of both program input and the time stamp of the program, as the behavior may be dynamic. Thus, adapting such studies to a co-design framework, where the program may change, is a difficult task. Our work aims to be automated, meaning we can create the low-fidelity components of our multifidelity models on the fly, without needing to perform a new study each time we add a new benchmark to the suite or change the benchmarks.

2.3 Multifidelity modeling

So far we have looked at related work in the field of computer architecture, but there is related literature on multifidelity simulation the world of physics simulations that is somewhat unconnected to our area of study. However, their methods are quite relevant and we share many of their ideas here. We will focus on *multifidelity simulation* which is defined as either a simulation which uses data from multiple levels of fidelity to construct a single aggregate model, or a simulation which uses different levels of fidelity at different points in the simulation.

The most relevant paper [5] develops a framework for adapting simulation to be multifidelity. Largely, we agree on the tasks involved in adapting a high-fidelity model with low-fidelity surrogates. Namely, it is important to determine low-fidelity candidate models and to identify where one might switch between high and low fidelity models. However, there are some key differences between our work and theirs:

(1) They focus on outer-loop problems, in which a single model is repeatedly evaluated with varying parameters, meaning simulations are independent of each other. Our work, however, is interested in the acceleration of a single simulation with changing fidelity.
(2) They are able to determine ahead of time where their regions of interest are, which is the region they need high-fidelity simulation for, meaning their technique is offline.

2.4 Summary

We have covered the most relevant related work in this section, but the field is quite large so those wishing to explore further should turn to more comprehensive treatments of the topic [8, 10].

3 MULTIFIDELITY COMPONENTS

Before we take a look at the new components we need to add to our simulation, let us first take a look at the system we will be simulating. For now we will take a high-level look, and later in the experimental results we can examine the finer details.

3.1 Simulated architecture

For this study, we have chosen a simple architecture with a trace-based core model, a two-level cache, backed by main memory. This is depicted in Figure 2.

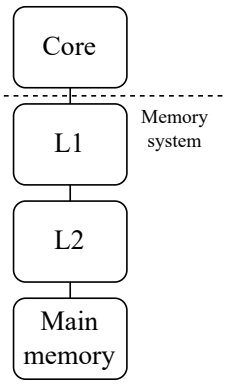

Figure 2: Simplified diagram of the simulated node architecture. This work will create a multifidelity memory system model which accelerates the components below the dashed line.

The simulator is trace-driven, meaning the core model will simply serve to track the number of outstanding transactions. The memory system model, which is everything in the figure except for the core model, will serve as a timing simulator, meaning it only needs to figure out how long each memory access takes; it does not need to keep track of actual data being read from or written to memory.

With the basic structure intact, we can discuss the components we added to this simulation to (1) decompose the program into homogeneous regions called phases, (2) find representative regions of those phases, and (3) create low-fidelity models from those representative regions.

3.2 Phase detection

The first problem we must address in creating a multifidelity simulation is decomposing our problem domain, both in time and in space. In a physical simulation, how space is decomposed might change throughout the simulation, such as eddies moving around in a fluid simulation. Fortunately for us, computer architecture simulations have well defined boundaries between components, so changing the fidelity of a single component is no issue, as long as it maintains the interfaces it uses to talk to other components. However, we still want to decompose the problem in time as well; this is where phase detection comes in.

In Figure 3 we can see an example of the data that we want to replicate with our model. This figure shows the mean memory access latency for each interval of 75,000 instructions in `correlation`, a benchmark in PolyBench, a benchmark suite of simple kernels used by compiler writers [13]. We can see that this program goes through different phases of behavior which would be difficult to capture in a simple model that aims to summarize the entire program. Phase detection will help us to isolate these regions of distinct behavior.

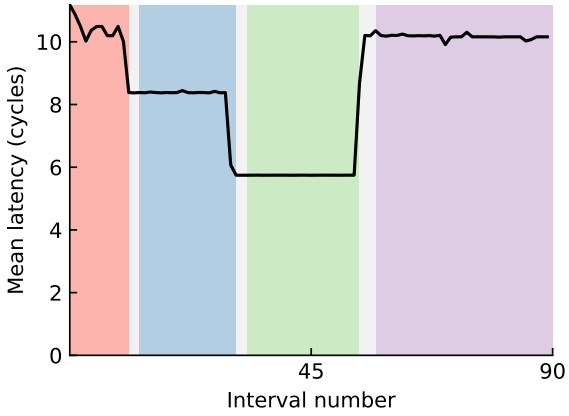

Figure 3: Mean memory access latency as a function of time for the *correlation* benchmark. Each point represents an average over 75,000 references. Background colors indicate different phases identified by the phase detector. Grey regions are *transition phases*, which indicate that the instruction working set was changing too quickly at that point to declare a new phase.

Phase detection has been used in an *offline* fashion by the popular SimPoint [24] technique to cluster together similar regions of execution so that researchers can simulate a single representative region from each cluster, instead of needing to simulate the entire program. Our multifidelity algorithm differs significantly though, as it is *online*. This led us to choose the algorithm from Dhodapkar et al.[7], which was used in prior multifidelity work as well [14]. Readers should see Appendix A of that paper to see the algorithm in more detail.

3.2.1 Dhodapkar algorithm. The algorithm we chose for phase detection uses only the stream of instruction pointers to determine the current program phase. It functions, at a high level, as follows:

(1) Break the stream of instruction pointers (IPs) into uniformly sized, non-overlapping windows
(2) Assign each interval a *signature* by hashing each IP into a bit vector
(3) Compare the signature to previous intervals using a distance function and a threshold. If a few intervals in a row

are similar, meaning their distance is below the threshold, we can say we are in a phase.

As this algorithm relies only on IPs, it is unable to detect data-dependent changes in behavior. For instance, a program like Spatter [15] has memory access strides that change based on user settings, even though the kernel being executed, and thus the stream of IPs, does not change. This means that this particular phase detection algorithm would not work well for programs with that behavior. We'll discuss potential improvements to phase detection at the end of the paper in Section 8.

3.3 Stability detection

Armed with phase detection to decompose our program in time, we now want to create a low-fidelity model for each phase that we can use in place of the full memory system simulation. However, we need to decide what data we should use to train such a model. Furthermore, this is happening online, so we need to make the decision of when we should stop collecting data from the phase so that we can switch to the low-fidelity model. Prior work simply chose the first few intervals of a phase, however, this will not work in this implementation as the first few intervals of a phase often have very different behavior as the cache is warming up. For this work, we develop *Stability Detection*, which is going to tell us when the distribution of latencies of memory references has stabilized, and give us a representative region of that phase.

We will borrow the *Ft-Pj-RG method* from the world of Monte Carlo simulation.

3.3.1 Ft-Pj-RG Algorithm. The Ft-Pj-RG algorithm was developed for use in Monte Carlo simulations, where users want to run many simulations and wait for some statistic of interest to stabilize, letting them know they can stop running simulations. We have a similar situation, where many phases in a program display high variability in the latency at the beginning of a phase as the cache warms up, and then get to a steady state. Thus, we will use this algorithm to help us find when that steady state has begun. We will describe the high-level aspects below, but readers can find the full algorithm in the original paper [20].

The algorithm works by creating two consecutive windows of data, and running a series of statistical tests on them. First an F-test is run, to test for equal variance of the two windows. If this passes, a t-test is run to test if the windows have the same mean. Finally, if this passes, a series of projection tests are run to see if the data in our windows is predictive of some number of future windows. If any of tests fail, the algorithm advances the windows, grows them, and returns to the F-test. Once all tests pass, we declare that we have found a stable region and use that data to train a low-fidelity model which we will explain in the next subsection, Section 3.4.

We had to make one change to the algorithm, which was that our data, the latency of memory accesses, can be quite noisy on a small scale. Thus, we average together a number of points to form a single data point that is used as input for this algorithm. How exactly we chose how many points to aggregate together, as well as all the other parameters for this algorithm, is explained in Section 5.

3.4 Low-fidelity memory system model

Our low-fidelity model only needs to do one thing: model the latency of a single memory access. Normally, a read or a write would be sent to the memory system and it would return some time later to the core. Our model will speed this up by predicting the latency of the access and immediately scheduling it to arrive at the core some number of cycles in the future.

For this work, we will evaluate a single low-fidelity model - a fixed latency model. This means that for every phase in which the stability detector finds a stable region, we will take the mean latency of the stable region and use that value as the latency of all future references in that phase instead of sending them to the memory system simulator. In practice we can't set a delay at a finer granularity than 1 cycle, so for each memory access, we assign it a latency of ceil(G) with probability $? \equiv G - \text{floor}(G)$—a value which always lies in $[0, 1)$—and otherwise assign it a latency of floor(G) with probability $@ \equiv 1 - ? = 1 - G + \text{floor}(G) = \text{ceil}(G) - G$. The expected value of the assigned latency is, therefore, G, as desired.

3.5 Multifidelity system

Now that we have our components, we can take a look at where they fit into our simulation, depicted in Figure 4. The first component, the phase detector, needs access to the instruction pointers, so it is placed with the core model. While instruction pointers could be shared with the rest of the system, placing the phase detector here means that the phase detection algorithm can easily be changed in the future to one that uses other execution information, such as the number of branch mispredictions.

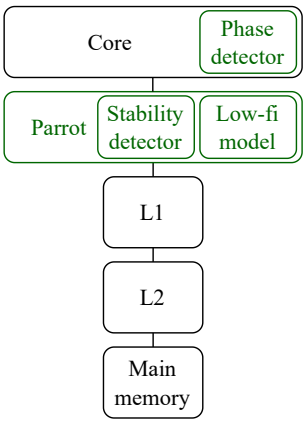

Figure 4: The updated simulation model. We have added the components in green, which are the phase detector, and the Parrot, which itself includes the stability detector and the low-fidelity model.

The other two components are placed within a new component which we call the Parrot, as this is the component that will try to mimic the behavior of the memory system with a statistical model[2]. Placing the Parrot at the top of the memory hierarchy means it can observe the memory latency behavior in each phase so that

[2]To avoid introducing delay to each memory access, the Parrot component is clocked at twice the frequency of the rest of the simulation.

the stability detector can determine when it has collected enough information. Once the stability detector has found a stable region for a phase, it can create a new low-fidelity model for that phase and use it to accelerate simulation.

With all of our multifidelity components in place, we are ready to look at the full multifidelity algorithm.

4 MULTIFIDELITY ALGORITHM

Now that we have phase detection, stability detection, and a low-fidelity model, we have everything we need for the full algorithm, save the phase detection and stability detection parameters, which we will discuss in the following section.

At the beginning of the program, we begin running the phase detector, which will send messages to the Parrot component when phases begin and end. The rest of the algorithm can be understood on a per-phase basis. When a new phase is identified:

(1) Begin running the stability detector after enough data has been collected
(2) If a stable region is found, create a low-fidelity model and use that for the rest of the phase and for future invocations of the phase
(3) If a stable region is not found, wait to collect more data from the phase
(4) If the phase ends before a stable region is found, give up on searching for a stable region for that phase

This algorithm is online, as phases and stable regions are identified during simulation, and low-fidelity models are used whenever possible. While there are scenarios where stability won't be found, meaning we can't use our low-fidelity model, this is by design, as it prevents us from using the simpler models where they are not appropriate; we only want to speed up the areas of execution displaying behavior which can be faithfully represented by our fixed-latency model.

One aspect we have avoided so far is the selection of the parameters for the phase and stability detector, but we will discuss those next.

5 PHASE AND STABILITY PARAMETER SELECTION

As the phase and stability detectors are both parameterized, we need a way to choose appropriate settings. Ideally, we would find a single setting that worked well for any combination of benchmarks and architectures or a way to adjust these parameters online, but that is beyond the scope of this work and is left as an open problem. For this study, we have collected data from regular runs of the simulator (with no multifidelity behavior enabled) and run it through an offline optimization procedure so that we can determine appropriate settings for these parameters. For our purposes, we will adjust three of the phase detection parameters and five of the stability detection parameters. See tables 1 and 2 respectively. We ignore the majority of the Ft-Pj-RG parameters in our search as the authors of the method note that the projection test is the most domain-dependent aspect of the algorithm. The default values are used for the F- and t-tests, and are listed in the original paper on the technique. For the rest of this section, an assignment of values to those eight parameters will be referred to as a *design point*.

Parameter	Explanation
`stable_min`	The number of consecutive similar phases needed to declare a new phase
`threshold`	The maximum distance between two signatures to consider them similar
`interval_len`	The number of instructions in an interval

Table 1: Phase detection parameters

Parameter	Explanation
`summarize`	The number of latencies averaged together to form a single data point
`window_start`	The initial size of a window. This changes throughout the algorithm.
`proj_dist`	How far out the projection test looks
`proj_delta`	How close the projection must be to the real point for a test success
`p_j`	How many projection tests must succeed for the test to pass

Table 2: Stability detection parameters

The primary focus of our evaluation will be on the PolyBench benchmark suite [13], so we will explain the parameter search with those benchmarks in mind, although this process was repeated for the other benchmark suites as well. PolyBench was chosen as it contains a large number of simple kernels which are easy to simulate and have easily understood phases, as the code is quite short. Thus, the latency plots in Figure 6 that we will look at later can be interpreted much more easily. The length of the kernels also makes running numerous simulations easier. They do limit us, however, to single-threaded experiments.

To begin, we collected 5 traces of each of the 30 benchmarks run on the medium inputs in PolyBench. Multiple traces are required as Ariel grabs the traces from running programs, so each is slightly different. Using multiple traces of each benchmark allows us to ensure we choose parameters that are robust to small changes in the input. We collect both the instruction pointer (IP) for every access, as this is needed by the phase detector, and the latency of each access, as this is needed by the stability detector and by our low-fidelity model.

It would take prohibitively long to run many settings of our multifidelity model, so we need a way to estimate the performance of each design point offline. We are concerned two different metrics in this work: speedup and accuracy. To achieve speedup, we will want our phase detector and stability detector to quickly produce stable regions so that we may begin using low-fidelity models early in the program. To achieve accuracy, we want our detectors to give us good quality stable regions that match closely with the behavior of the rest of the phase. As such, we developed two metrics to estimate these values from the recorded traces for a given design point.

5.1 Estimating Speedup

We can estimate speedup by looking at the amount of time we could potentially spend in low-fidelity models given how much time we spend in phases after stable regions have been identified. We collected memory traces for the PolyBench benchmarks and estimated speedup of a single trace as follows:

$$\text{Speedup}_{\text{est}} = \frac{!}{\sum_{\delta \in \text{phases}} (;_\delta - B_\delta)}$$

where $!$ is the length of the trace, $;_\delta$ is the length of phase δ and B_δ is how long it takes to find a stable region for phase δ. B_δ is less than or equal to $;_\delta$. This means that a configuration that spends more time in a phases where stability is quickly identified will have a higher estimated speedup. Were this our only metric, we would end up with configurations that have loose requirements for interval similarity and stability detection, but not necessarily configurations that give us good stable regions. Thus, we have an accuracy metric as well.

5.2 Estimating Error

We can estimate the error of a configuration by calculating how similar the stable regions identified are to the rest of the phases they come from. As our low-fidelity model predicts the latencies based on the mean latency of the stable region, we compare the mean of the stable region to the mean of the entire phase to judge accuracy. Formally, the estimated error of a configuration is:

$$\text{Error}_{\text{est}} = \tilde{\sum_{\delta \in \text{phases}}} \frac{G_\delta - G_\delta}{G_\delta} * \frac{;_\delta - B_\delta}{!}$$

where $!$ is the trace length, $;_\delta$ is the length of phase δ, B_δ is how long it takes to find the stable region in δ, G_δ is the mean latency of phase δ, G is the mean latency of stable region δ if it has one, or G_δ otherwise. This essentially average percent error for between the mean latency of each stable region and the phase phase it comes from, weighted by time spent in that phase after the stable region is identified. A perfect score is 1. [3] Were we to use this criteria in isolation, we would end up with configurations that essentially never detect stability, as there will always be some amount of error when deciding to pick a stable region. Thus, we must optimize for both metrics simultaneously.

5.3 Parameter space exploration

We swept over roughly 6.6 million parameter combinations, as shown in Table 3. Each combination was evaluated on each of the 5 traces of each of the 30 benchmarks. To choose a single configuration to use in the multifidelity simulations, one option would be to combine the scores for the benchmarks, weighting them appropriately based on the length of each. However, this led to configurations which were somewhat fragile, meaning they would find a stable region for some runs of a benchmark, and sometimes they would not. Thus, we chose a different strategy.

To pick a single configuration, we first down selected by looking only at settings we deemed *robust*, meaning that if they produced multifidelity behavior in some benchmark on one trace, they produced multifidelity behavior for that benchmark for *all five* traces. Here, *producing multifidelity behavior* means that the configuration identified at least a single stable region in some phase. Then, we selected only configurations which estimated less than 2% error, and picked the one with the highest predicted speedup for each benchmark. We found that when looking at the configurations we were left with, there was a small range of values for each parameter. For parameters with more than one value at this point, we picked the one that worked well for the most benchmarks.

In Figure 5, you can see the entire parameter space we needed to select from. Each plot shows the configurations for each benchmark, with estimated speedup on the x-axis and estimated error on the y-axis. Thus, the bottom right corner is most optimal. The red dots show the Pareto optimal points. We can see that some benchmarks with very simple behavior such as `2mm` are very easy to optimize for as we can get very close to the bottom right of the plot. However, for others such as `durbin`, it can be difficult to decide what the best trade-off between accuracy and speedup is. Examining the latency trace in Figure 6 for this plot shows that the latency is not stable, and thus our fixed-latency model is not a good fit so the only way to get better accuracy is to use more restrictive stability detection parameters, which limits speedup. Refining and automating this procedure is left to future work.

In Figure 6, you can see the output of the phase detector for each benchmark for the configuration we selected. We see that a number of benchmarks have a single phase, but that there is often complex behavior within those phases, that may not work well for our model. This is what the stability detector aims to correct for. Some benchmarks have seemingly simple behavior but end up with very short fine-grained phases that don't work well with the stability detector such as `jacobi2d` and `lu`.

This method of choosing parameters relies on some simulations run in full-detail. In Section 7, we will describe when we had to repeat this technique and when we found we could share parameters between simulator configurations. In the future, we hope to simplify this process so that full-detail simulations are not required. We also expect that these benchmarks will experience better speedup with a more advanced technique, which will discuss in Section 8.

Parameter	Explanation
`stable_min`	[**3**, 4, 5]
`threshold`	[0.5, 0.6, **0.65**]
`interval_len`	[10,000, 50,000, **100,000**]
`summarize`	[20, 25, 50, **75**, 100]
`window_start`	[1000, **1500**, 2000]
`proj_dist`	[5, **10**, 15, 20]
`proj_delta`	[0.25, 0.5, 1.0, **2.0**, 2.5, 3.0]
`p_j`	[4, 6, 8, **10**]

Table 3: Parameter search space. The configuration chosen for multifidelity PolyBench simulations is shown in bold.

[3] The weights (second multiplicand) will not sum to one. This is fine because intervals simulated in high-fidelity, i.e. intervals belonging to stable regions or intervals belonging to no phase, have zero error.

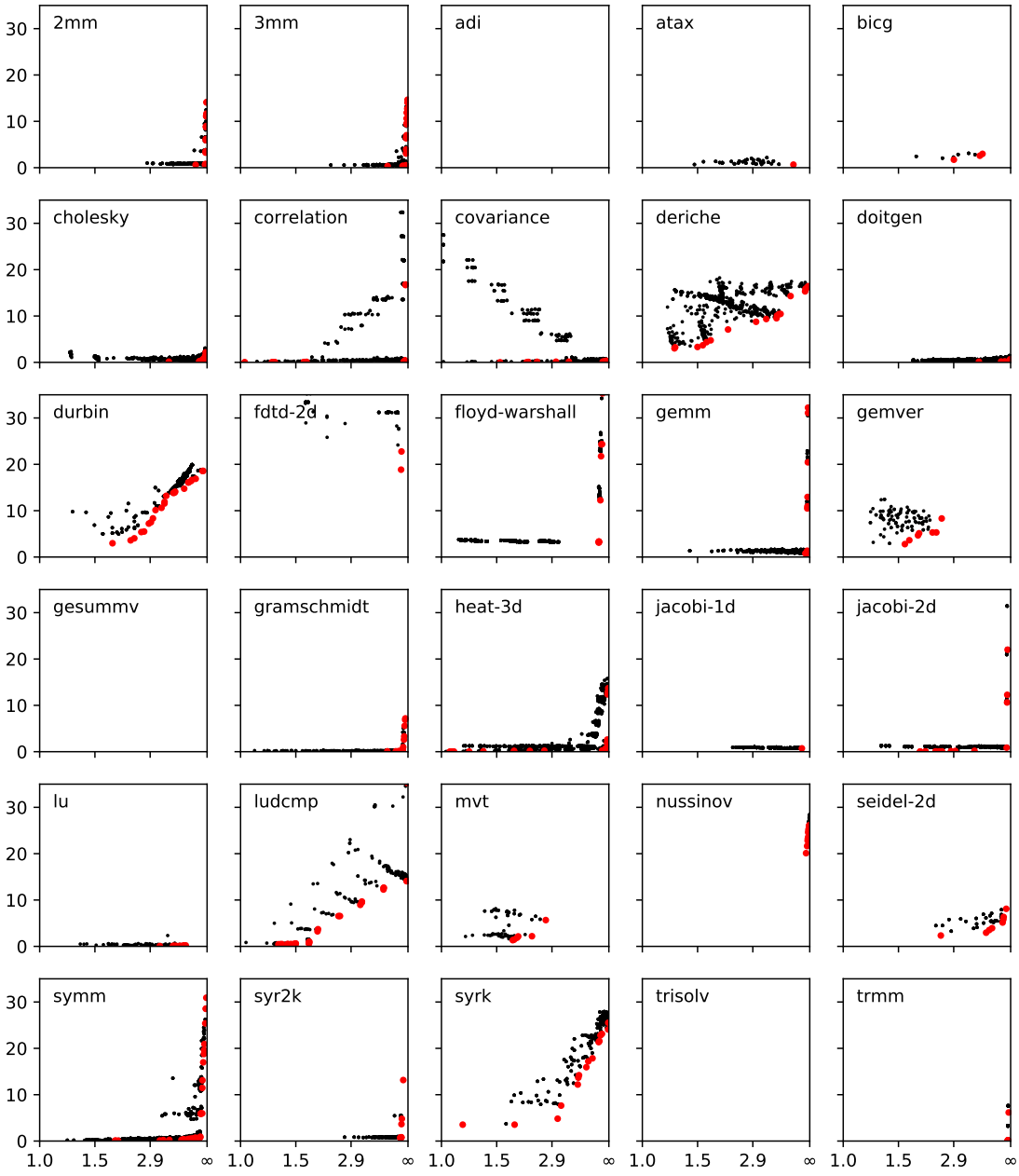

Figure 5: Phase detector/stability detector parameter search. Each point in each small plot is an an average score across 5 runs of a benchmark. The score on the x-axis is estimated speedup (defined in Section 5.1) and the score on the y-axis is estimated error (defined in Section 5.2). The best score would be in the lower right corner of the plot, meaning a high speedup but a low error. The Pareto front is shown in red.

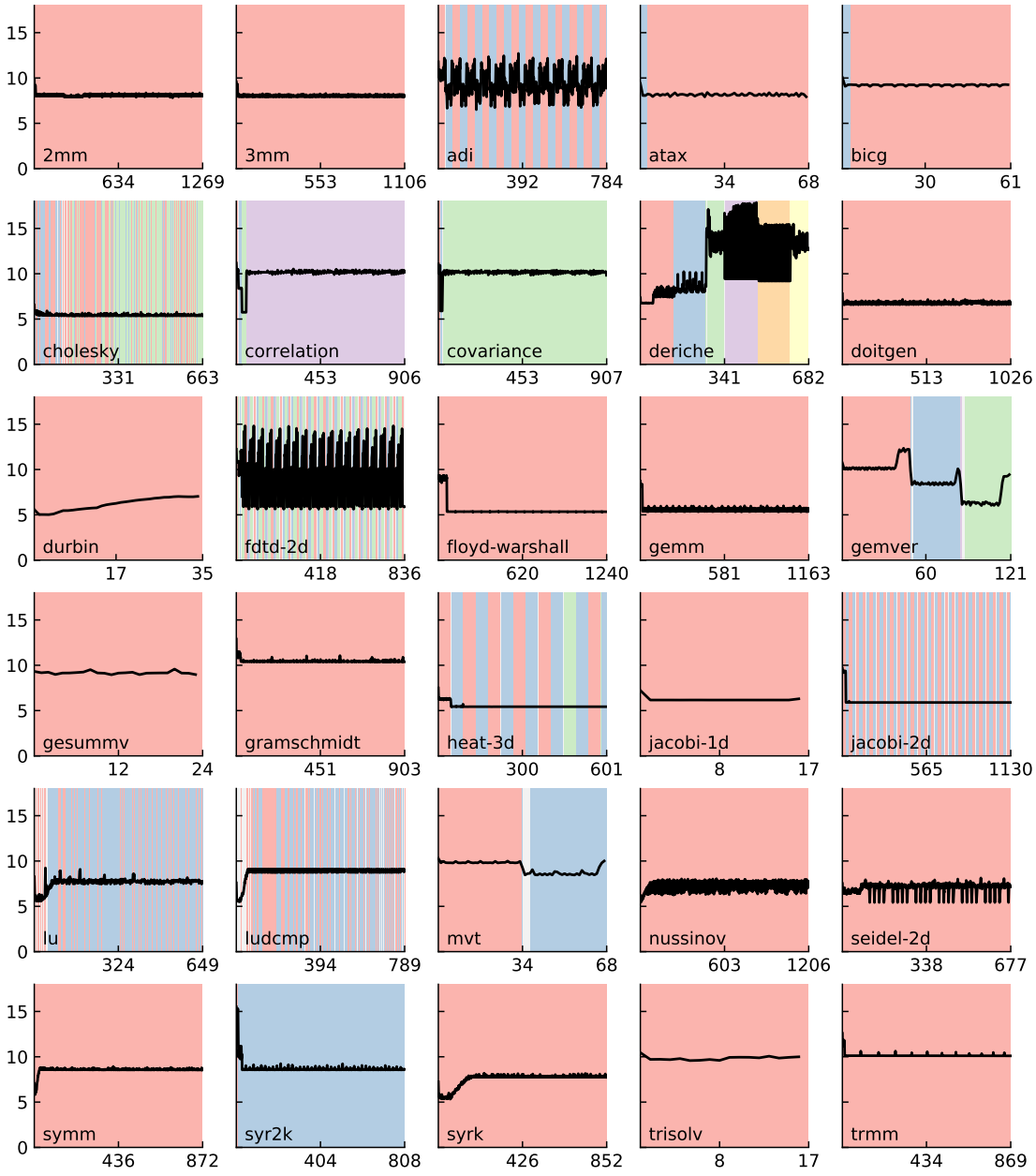

Figure 6: Mean memory access latency per 75,000 instruction interval. The background of each plot denotes the program phase, as determined by the phase detector.

6 EXPERIMENTAL DESIGN

In our evaluation, we are interested in two metrics, error and speedup. Error will be measured by the percent error in the simulated instructions per cycle (IPC), as this is a metric that is not lost when we replace the detailed memory simulation with our low-fidelity model, as opposed to something like the number of cache hits. We will be comparing multifidelity simulations with *normal* simulations, which is just the same simulator configuration with no phase detection or Parrot component. While our simulations are traced-based, we will see in this section that the traces are generated from running programs by the Ariel component. This means that each trace is a little bit different, so the simulated IPC for each program is a random variable. Thus, both normal and multifidelity runs have to be repeated to get bounds on the simulated IPC. Let us now take a closer look at the configuration of the simulated system.

6.1 SST Components

The overall structure of our simulator was described in Section 3.5. We have implemented this system in the Structural Simulation Toolkit (SST) [25], which is a collection of simulation components and interfaces, so we must decide which models we use and how they are connected together. Table 4 lists all the parameters we used for our simulation components.

6.1.1 Core model. We use `Ariel` as our core model, which is a Pin-based trace-driven model. Pin is a binary instrumentation tool which allows users to inspect and modify executing binaries [18]. Ariel uses Pin to collect the memory references of running programs and send those references to simulated cores, which then pass them on to the memory system simulator. Ariel's core model is very simple; it does not track dependencies between instructions, it only keeps track of how many outstanding references there are, and issues a maximum number of references per cycle. It is primarily used as a lightweight frontend for memory simulators. For all the experiments in this paper, Ariel will be scoped to only collect memory references for the kernel under study, which greatly reduced variability between runs.

6.1.2 Caches. We use the standard cache model for our simulation, which is `memHierarchy.Cache`.

6.1.3 Main memory. For main memory, we use SST's memory controller `memHierarchy.MemController`, which connects to a backend to provide detailed timing simulation for different types of DRAM. For the backend, our default will be DRAMSim3 [17], but we will also run experiments with a simpler simulator that is included with SST, `memHierarchy.timingDRAM`. DRAMSim3 supports a number of different DRAM models, so we picked the 8GiB HBM2 model for our experiments as it is a modern technology. This model is referred to by DRAMSim3 as `HBM2_8Gb_x128.ini`. For the `timingDRAM` experiments, we used the default settings with a size of 8GiB.

6.2 Host system

Our experiments were conducted on a server with an Intel Xeon Gold 6338, a 32-core, 64-thread processor built on the Ice Lake architecture with 512 GB of RAM. SST runs in a single process and the traced program runs in a separate process. Thus, the two can run in parallel, but SST still spends a lot of time waiting for instructions to be available from the traced process. However, as our work only concerns the memory system simulation, that parallelism will not affect our timings.

Component	Parameter	Value
Core	Cores	1
	Outstanding transactions	16
	Frequency	2.0 GHz
	Issue rate	1
L1 Cache	Frequency	2.0 GHz
	Size	32 KiB
	Line size	64 bytes
	Access latency	2 cycles
	Coherence	MESI
	Replacement	LRU
	Associativity	4
	Banks	8
L2 Cache	Frequency	1.0 GHz
	Size	1 MiB
	Line size	64 bytes
	Access latency	20 cycles
	MSHR latency	5 cycles
	Coherence	MESI
	Replacement	LRU
	Associativity	4
	Banks	8
Main Memory	Size	8GiB
Parrot	Frequency	4.0 GHz

Table 4: Simulator parameters. Neither cache uses prefetching. All links have a latency of 100ps, except those connecting the Parrot component which have a latency of 50ps.

6.3 Benchmarks

The first benchmark suite we will look at is PolyBench, a set of 30 simple kernels used by compiler writers for the evaluation of polyhedral optimizations [13]. They are useful for evaluating phase detection as they display a variety of phase behaviors, which provides additional variability for our system to detect and evaluate when it should and should not use a low-fidelity model.

Before we run our benchmarks, we will take a look at how well we can expect to do, given our simulator design and our fixed-latency low-fidelity model.

6.4 Potential speedup

As our model only affects the memory system, the speedup we can achieve is limited by how much time our simulation actually spends in the cache and main memory models.

We ran each benchmark in the original simulator without any of the multifidelity components. Each was run for a maximum of 100 simulated milliseconds. We were able to trace the amount of time

spent in each SST component with their built-in profiler. If SST-Core is compiled with `--enable-profile`, then each component can be timed when invoking `sst`.

For these runs, Table 5 shows how much time is spent in the core model and in the memory system simulation. A simple Amdahl's law calculation shows us that in the best case, we can achieve an average speedup of 2.58 speedup for the entire benchmark suite. While not shown in this chart, there is some overhead imposed by the Parrot component. Taking this into account, the maximum possible speedup we can achieve with the current Parrot implementation is 2.17. The data for individual benchmarks is included in Table 5.

benchmark	t_{Ariel}	t_{memH}	t_{Parrot}	MPS	A-MPS
2mm	2418.17	4007.74	465.62	2.66	2.23
3mm	2125.32	3604.09	417.44	2.70	2.25
adi	1489.42	2260.56	275.69	2.52	2.12
atax	71.03	109.22	13.80	2.54	2.12
bicg	69.43	100.47	13.11	2.45	2.06
cholesky	1296.58	1742.86	236.74	2.34	1.98
correlation	1877.55	3278.77	334.69	2.75	2.33
covariance	1852.30	3196.09	325.89	2.73	2.32
deriche	800.28	1182.60	142.24	2.48	2.10
doitgen	1944.98	2904.72	355.72	2.49	2.11
durbin	34.00	58.04	7.05	2.71	2.24
fdtd-2d	1578.39	2296.50	296.15	2.45	2.07
floyd-warshall	2259.06	3122.27	430.24	2.38	2.00
gemm	2132.90	2695.64	384.02	2.26	1.92
gemver	132.38	209.96	25.96	2.59	2.16
gesummv	26.05	39.97	5.02	2.53	2.12
gramschmidt	1739.03	3300.87	330.91	2.90	2.43
heat-3d	1295.96	1550.21	217.31	2.20	1.88
jacobi-1d	18.99	28.18	3.53	2.48	2.09
jacobi-2d	2144.12	3135.47	415.04	2.46	2.06
lu	1300.37	2196.92	250.44	2.69	2.26
ludcmp	1551.79	2854.33	304.74	2.84	2.37
mvt	72.94	125.01	14.53	2.71	2.26
nussinov	2327.72	3796.01	450.41	2.63	2.20
seidel-2d	1308.19	1797.04	246.35	2.37	2.00
symm	1690.12	3023.76	331.05	2.79	2.33
syr2k	1619.18	2855.17	314.99	2.76	2.31
syrk	1626.11	2702.28	308.13	2.66	2.24
trisolv	18.10	28.98	3.55	2.60	2.17
trmm	1372.07	2554.15	262.74	2.86	2.40
Mean	1273.08	2025.26	239.44	2.58	2.17

Table 5: Maximum potential speedup of multifidelity memory system simulation. PolyBench medium inputs simulated for a maximum of 100ms. The time spend in the core, the memory system, and the Parrot are given in seconds. MPS is the maximum potential speedup without considering the overhead imposed by the Parrot, and A-MPS is the adjusted maximum given the current Parrot implementation.

6.5 Accuracy estimates

Before we begin, we want to ensure that we are improving on simpler methods. For this, we will compare against a theoretical method that knows the mean memory latency access for the program before it begins, and uses that as the low-fidelity for the entire simulation, with no detailed memory simulation whatsoever. We will calculate how much error we expect from such a method by examining traces collected from our simulation instead of implementing this technique.

To create an error number for this theoretical fixed-latency method, we calculate the mean absolute percent error, or MAPE, as follows:

$$\mathrm{MAPE}_{\mathrm{fixed}} = \sum_{\delta=1}^{\#} \frac{G - ;_{\delta}}{G} * \frac{100}{\#}$$

where G is the mean latency for the program, $;_{\delta}$ is the latency of access δ, and $\#$ is the number of accesses in the program. To calculate an error number for our multifidelity simulation, we use a trace from a regular simulation, but we calculate what phases and stable regions would be identified by using the same parameters as we have chosen for multifidelity simulation. We then calculate a MAPE value for each phase by comparing the mean of the stable region to the accesses after the stable region, as these are the only accesses that would use that value in a multifidelity simulation. The rest of the values have zero error, as they would be run through the actual memory system simulator. As you can see in Table 6, we improve upon the error in the vast majority of cases. Importantly, even though our approach has less information, it is able to improve upon the error by better matching the dynamic behavior of the program under study. These error numbers may look large, but we will see in the next section that we still achieve high accuracy in the overall IPC of the simulation. A number of benchmarks have zero estimated error, meaning they are not predicted to have any stable regions that we can create low-fidelitymodels from.

7 RESULTS

We ran experiments on the PolyBench benchmark suite as well as matrix multiply and Spatter. The results are summarized at the end of this section in Table 7.

7.1 PolyBench

Our first three experiments were conducted with the 30 benchmarks in PolyBench. We ran each benchmark 5 times with and without the multifidelity components enabled.

7.1.1 Initial experiments. Using the methodology described in Section 5.2, we selected parameters for phase detector and stability detector. Thus, we used the five normal runs of the benchmarks to find suitable parameters for the multifidelity runs. The error and speedup results are in Figure 7. As we can see from the speedup plots, 16 of the benchmarks did not speedup, as the stability detector did not find any stable regions to create low-fidelity models from. Thus, the overall average speedup is limited to 1.46 for this set of experiments. We do, however, achieve a high degree of accuracy, with a MAPE of only 1.98% in the simulated IPC. The numeric values of these plots can be seen Table 8 in Appendix B.

7.1.2 System modifications. We wished to examine the generality of the parameters we had found in the first set of experiments, so we change our model in two different ways: changing the issue rate of the core and changing the DRAM model. When changing

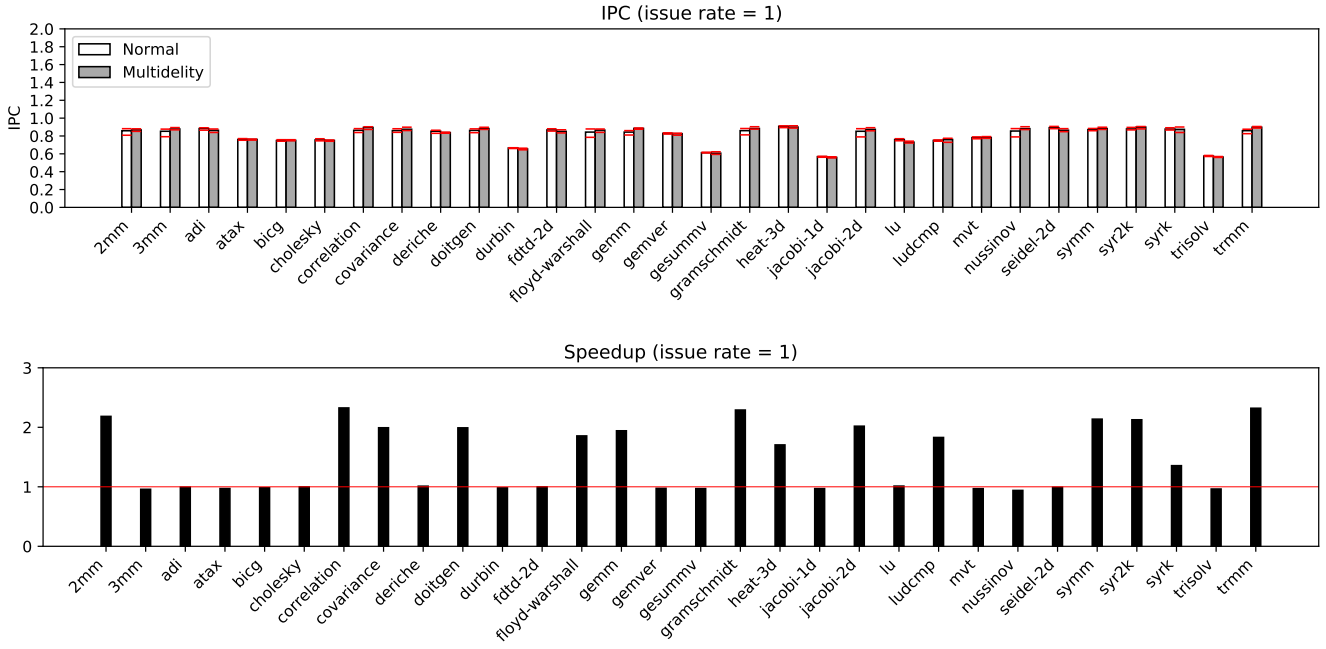

Figure 7: Accuracy and speedup results for PolyBench with a core issue rate of 1 (top and bottom, respectively). The red dashes in the accuracy plot represent the minimum and maximum of the five runs, and the bar itself is the mean. In the speedup plot, the red line is at 1; benchmarks near this line did not undergo multifidelity behavior.

the issue rate of the core, we found that while the variability of the multifidelity runs increased, we still achieve a comparable speedup of 1.47 and MAPE of 4.67% in the IPC.

We also examined changing the model of the DRAM from the HBM2 model from DRAMSim3 to the timingDRAM model included with SST while retaining our original single issue core. Overall, this led to a 8% average difference in the IPC of the normal simulations, so the effect of the DRAM model is considerable. We found that we achieved slightly better average speedup in this case, at 1.59, likely due to the fact that SST is able to disable the timingDRAM's clock when it is not in use, which is not true of the DRAMSim3 backend. The MAPE of the IPC for these runs is 1.50.

The full results for these two experiments are in Tables 9 and 10 in Appendix B.

7.2 Other benchmarks

To demonstrate the usage of multifidelity simulation for microarchitectural studies, we perform a matrix multiply cache blocking study and take a look at the Spatter uniform stride inputs.

7.2.1 Matrix multiply. To continue our evaluation, we wished to examine how the method would work for the purposes of a microarchitectural study, such as examining the effect of cache blocking on the overall IPC. We ran a basic matrix multiply (= ∗) on 180x180 matrices with varying block sizes. We re-ran the optimization procedure to find new settings for the phase detector and the stability detector. Due to the simplicity of the behavior, we found that a looser constraint on the projection test in the stability detector allowed for slightly more speedup. The optimal phase detection parameters remained unchanged. The experiments achieved an average speedup of nearly 2x and very low error of only 0.78%. However, we found that even with the small error, our multifidelity simulations produced an IPC curve that was too noisy to estimate the optimal block size from. We believe that this architecture was not ideal for demonstrating the usage of matrix blocking as the performance of different block sizes was very tightly grouped. Thus, we also examined a benchmark which better demonstrated the performance aspects of this architecture.

7.2.2 Spa er. Our final evaluation is on the uniform stride benchmark suite from Spatter. Spatter is an irregular memory access benchmark that can be configured to run many different memory access patterns, either created by hand or generated from traces of other benchmarks [15]. For our purposes, we will use only a simple input, the uniform stride gather input. The uniform stride pattern is quite simple: a stride-1 gather is akin to STREAM [19], reading every element of an array, except it only performs reads, not writes. A stride-N gather will read every N^{th} element of an array. An element here is an 8 byte double. Each kernel will perform a total of 128,000 reads.

We ran strides 1 through 8, and achieved an average error of 2.45% with a speedup of 1.33. In Figure 8, we see that except for an issue with the stride-1 simulation, we faithfully represent the slope of the regular simulations. The modest speedup is due to the fact that the Spatter runs are quite short and so we do not spend as much time using the low-fidelity model. Limitations in the current implementation of the benchmark prevented us from extending

Benchmark	$MAPE_{58G43}$	$MAPE_{<5}$
2mm	59.62	55.75
3mm	60.86	65.81
adi	86.89	0.00
atax	63.57	0.00
bicg	79.63	0.00
cholesky	9.13	7.67
correlation	99.25	94.14
covariance	99.21	90.66
deriche	97.97	0.00
doitgen	36.83	34.84
durbin	25.89	14.03
fdtd-2d	75.49	0.00
floyd-warshall	11.47	7.20
gemm	9.48	7.84
gemver	72.56	0.00
gesummv	77.54	0.00
gramschmidt	101.35	96.18
heat-3d	10.61	16.57
jacobi-1d	26.36	20.02
jacobi-2d	20.35	17.95
lu	54.37	13.82
ludcmp	77.02	5.25
mvt	85.14	0.00
nussinov	51.10	0.00
seidel-2d	37.02	0.00
symm	68.11	63.24
syr2k	74.93	69.19
syrk	50.94	11.53
trisolv	95.65	0.00
trmm	100.03	97.86

Table 6: Estimated dynamic error for each of the PolyBench benchmarks. The $MAPE_{58G43}$ column represnts the mean error for each reference if we knew the mean latency for the program ahead of time. $MAPE_{<5}$ represents an estimated error for our multifidelity runs.

Experiment	MAPE	Speedup
single-issue	1.96%	1.46
double-issue	4.49%	1.47
timingDRAM	1.50%	1.59
Matrix multiply	0.78%	1.98
Spatter	2.45%	1.33

Table 7: Summary of results from Section 7. MAPE is the mean absolute percent error in the IPC between normal and multifidelity runs.

these runs, but for a longer running pattern they can achieve the same speedup we saw with matrix multiply. The interesting takeaway is that even though Spatter is an input-dependent program, we are able to automatically create models that reflect this behavior, without needing to do any analysis on how program input affects the simulated behavior.

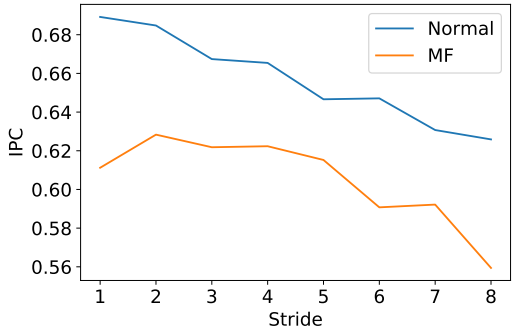

Figure 8: Spatter experiment results. On the x-axis, is the stride of the memory access pattern, and simulated IPC is on the y-axis. The overall mean error is 2.45%.

8 DISCUSSION AND FUTURE WORK

In this work, we presented 3 contributions: (1) a novel addition to the multifidelity algorithm, stability detection, (2) an implementation of components that can be re-used in other SST simulations, the phase detector and Parrot components, and (3) a new application of multifidelity methods to the memory system. We found that while the algorithm, in its current state can achieve reasonable speedup on a number of benchmarks, the overall speedup is limited by benchmarks that we could not identify stable regions for with our stability detector. While we could have used more lenient stability detection parameters, we chose parameters that we estimated would keep the error around 2%. We found that while this level of accuracy was useful in the simulation of Spatter, it may be too high for some studies, as we found with matrix multiply. Thus, we believe that our technique is most useful in contexts where users would want an automated, simpler model, such as in the case of large-scale simulation or in co-design studies.

The area of multifidelity computer architecture simulation is new, meaning there are plenty of places to explore improvements, both in the accuracy of the technique and the possible speedup.

8.1 Accuracy

A limitation of the work in its current state is its inability to produce error bars on the accuracy of the simulation without knowledge of the ground truth. This limits its ability to fully replace traditional, non-multifidelity simulation. We believe there are two promising avenues to address this: statistical sampling and changes to the phase detection algorithm.

As we mentioned in Section 2, some approaches use statistical sampling to produce error bars on the final IPC. This technique could be combined with a multifidelity algorithm to apply sampling per-phase. The combination of the two would give us the ability to detect complex phases and keep them from being simulated in low-fidelity, potentially improving the accuracy of sampled simulation.

The other approach would be to change the phase detector to use signals more closely aligned with the behavior under study. For example, if the phase detector had locality information, then a phase change would indicate that a new low-fidelity memory model should be trained, as the old one is likely now incorrect. In our study,

we used the instruction pointer working set, so we were unable to detect these changes in programs where kernel behavior changes throughout, but we corrected for it with the stability detector. A stronger phase detector may obviate the need for stability detection altogether.

8.2 Speedup

Our technique will benefit as more simulators are integrated into the SST ecosystem. As more models become available, we will be able to use the techniques discussed here to choose when it may make sense to switch between them during simulation to get the speedup of the faster model while maintaining high accuracy. For example, while we used DRAMSim3 in this paper, there are other simulators such as DRAMSys4.0 that act as other levels of fidelity for the DRAM [9, 26]. However, there are other bottlenecks we must consider than the models themselves.

As we noted in Section 6, the maximum potential speedup we can achieve with a multifidelity memory system is around 2.5x, as we are bottlenecked by waiting on Ariel to generate traces. Thus, work on creating a multifidelity trace generation model has the potential to speed up simulation further. This would require researchers to find a way to disable and re-enable tracing, allowing the native program to run at full speed while the simulation used its own memory reference generator. Such a generator may use statistical techniques to create a simplified memory stream, such as a Spatter pattern.

Even with an accelerated memory reference generator, there will still be a fundamental bottleneck of needing to do something with every single instruction. To achieve an order of magnitude speedup, it may be necessary to introduce even simpler models, perhaps where sections are not simulated in any way, but where statistics are re-used from previous executions of those regions. This is similar to how ^{2}Sim and SimPoint work, as mentioned in Section 2. It would also sense to use our multifidelity memory system in conjunction with other simulation acceleration techniques, such as Pinballs [23] or statistical sampling.

We also believe this work can extend naturally to the world of network simulation. Networks already have a number of models available for them, such as models that incorporate congestion, or those that only model the number of hops each packet takes. A multifidelity network model could detect when the simpler model would be useful and switch between in response to network behavior, giving researchers the benefit of a faster model without needing to characterize network traffic themselves.

8.3 Conclusions

With further research into multifidelity simulations, we believe they will become an important part of continuing to scale computer system simulations. Currently, researchers must perform time consuming studies for each program they are interested in to characterize the node-level and network-level behavior, which limits the number of experiments that can be simulated. Multifidelity algorithms automate the process of creating faster models, meaning they will be vital in enabling us to simulate tomorrow's supercomptuers.

ACKNOWLEDGMENTS

The authors would like to thank the SST team for the help with this project, particularly Gwendolyn Voskuilen and Scott Hemmert for the help in developing the models.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

This research was supported in part through research infrastructure and services provided by the Rogues Gallery testbed [28] hosted by the Center for Research into Novel Computing Hierarchies (CRNCH) at Georgia Tech. The Rogues Gallery testbed is primarily supported by the National Science Foundation (NSF) under NSF Award Number #2016701. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s), and do not necessarily reflect those of the NSF.

REFERENCES

[1] Tae-Hyuk Ahn, Damian Dechev, Heshan Lin, Helgi Adalsteinsson, and Curtis L Janssen. 2011. Evaluating Performance Optimizations of Large-scale Genomic Sequence Search Applications using SST/macro.. In *SIMUL TECH*. 65–73.

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[3] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization (TACO)*, Article 5 (2014), 23 pages. https://doi.org/10.1145/2629677

[4] Trevor E Carlson, Wim Heirman, Kenzo Van Craeynest, and Lieven Eeckhout. 2014. Barrierpoint: Sampled simulation of multi-threaded applications. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2–12.

[5] Seon Han Choi, Kyung-Min Seo, and Tag Gon Kim. 2017. Accelerated simulation of discrete event dynamic systems via a multi-fidelity modeling framework. *Applied Sciences* 7, 10 (2017), 1056.

[6] T.M. Conte, M.A. Hirsch, and K.N. Menezes. 1996. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*. 468–477. https://doi.org/10.1109/ICCD.1996.563595

[7] Ashutosh S. Dhodapkar and James E. Smith. 2002. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture* (Anchorage, Alaska) *(ISCA '02)*. IEEE Computer Society, USA, 233–244.

[8] Lieven Eeckhout. 2010. *Computer architecture performance evaluation methods*. Morgan & Claypool Publishers.

[9] Johannes Feldmann, Kira Kraft, Lukas Steiner, Norbert Wehn, and Matthias Jung. 2020. Fast and Accurate DRAM Simulation: Can we Further Accelerate it?. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 364–369. https://doi.org/10.23919/DATE48585.2020.9116275

[10] Qi Guo, Tianshi Chen, Yunji Chen, and Franz Franchetti. 2016. Accelerating Architectural Simulation Via Statistical Techniques: A Survey. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 3 (March 2016), 433–446. https://doi.org/10.1109/TCAD.2015.2481796

[11] Simon David Hammond, Karl Scott Hemmert, Michael J Levenhagen, Arun F Rodrigues, and Gwendolyn Renae Voskuilen. 2015. *Ember: Reference Communication Patterns for Exascale.* Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[12] Mor Harchol-Balter. 2013. *Performance modeling and design of computer systems: queueing theory in action.* Cambridge University Press.

[13] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. Polybench: The first benchmark for polystores. In *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence: 10th TPC Technology Conference, TPCTC 2018, Rio de Janeiro, Brazil, August 27–31, 2018, Revised Selected Papers 10*. Springer, 24–41.

[14] Patrick Lavin, Jeffrey Young, Richard Vuduc, and Jonathan Beard. 2021. Online model swapping for architectural simulation. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*. 102–112.

[15] Patrick Lavin, Jeffrey Young, Richard Vuduc, Jason Riedy, Aaron Vose, and Daniel Ernst. 2020. Evaluating Gather and Scatter Performance on CPUs and GPUs. In *The International Symposium on Memory Systems*. 209–222.

[16] Wonbok Lee, Kimish Patel, and Massoud Pedram. 2006. B ^{2} Sim:: a fast micro-architecture simulator based on basic block characterization. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis - CODES+ISSS '06*. ACM Press, Seoul, Korea, 199. https://doi.org/10.1145/1176254.1176303

[17] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *IEEE Computer Architecture Letters* 19, 2 (July 2020), 106–109. https://doi.org/10.1109/LCA.2020.2973991

[18] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.* 40, 6 (jun 2005), 190–200. https://doi.org/10.1145/1064978.1065034

[19] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[20] Chris Nellis, Thomas Danielson, Aditya Savara, and Celine Hin. 2018. The ft-pj-rg method: an adjacent-rolling-windows based steady-state detection technique for application to kinetic monte carlo simulations. *Computer Physics Communications* 232 (2018), 124–138.

[21] Daiheng Ni. 2011. Multiscale modeling of traffic flow. *Mathematica Aeterna* 1, 1 (2011), 27–54.

[22] EPFL Parallel Systems Architecture Lab (PARSA). 2020. QFlex. https://qflex.epfl.ch

[23] Harish Patil and Trevor E Carlson. 2014. Pinballs: portable and shareable user-level checkpoints for reproducible analysis and simulation. In *Proceedings of the Workshop on Reproducible Research Methodologies (REPRODUCE)*, Vol. 2.

[24] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for Accurate and Efficient Simulation. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (San Diego, CA, USA) *(SIGMETRICS '03)*. Association for Computing Machinery, New York, NY, USA, 318–319. https://doi.org/10.1145/781027.781076

[25] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (March 2011), 37–42. https://doi.org/10.1145/1964218.1964225

[26] Lukas Steiner, Matthias Jung, Felipe S. Prado, Kirill Bykov, and Norbert Wehn. 2020. DRAMSys4.0: A Fast and Cycle-Accurate SystemC/TLM-Based DRAM Simulator. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Alex Orailoglu, Matthias Jung, and Marc Reichenbach (Eds.). Springer International Publishing, Cham, 110–126.

[27] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[28] Jeffrey S. Young, Jason Riedy, Thomas M. Conte, Vivek Sarkar, Prasanth Chatarasi, and Sriseshan Srikanth. 2019. Experimental Insights from the Rogues Gallery. In *2019 IEEE International Conference on Rebooting Computing (ICRC)*. 1–8. https://doi.org/10.1109/ICRC.2019.8914707

A AVAILABILITY

Our simulator is available on GitHub at this link: https://github.com/plavin/multifidelity/releases/tag/memsys23.

B EXPERIMENT DATA

Benchmark	IPC_{true}	IPC_{mf}	Pct. Error	Speedup
2mm	0″86 ± 0″03	0″87 ± 0″03	-1.04	2.18
3mm	0″85 ± 0″03	0″88 ± 0″03	-2.91	0.96
adi	0″88 ± 0″01	0″86 ± 0″01	2.27	0.99
atax	0″76 ± 0″01	0″76 ± 0″01	-0.11	0.97
bicg	0″75 ± 0″01	0″75 ± 0″01	-0.22	0.98
cholesky	0″76 ± 0″01	0″75 ± 0″01	0.75	1.00
correlation	0″86 ± 0″02	0″90 ± 0″02	-3.60	2.33
covariance	0″86 ± 0″01	0″88 ± 0″01	-1.73	1.99
deriche	0″85 ± 0″01	0″84 ± 0″01	1.81	1.01
doitgen	0″86 ± 0″02	0″88 ± 0″02	-2.32	1.99
durbin	0″67 ± 0″00	0″65 ± 0″00	1.71	0.98
fdtd-2d	0″87 ± 0″01	0″85 ± 0″01	1.90	1.00
floyd-warshall	0″84 ± 0″03	0″87 ± 0″03	-2.54	1.86
gemm	0″84 ± 0″02	0″88 ± 0″02	-4.68	1.94
gemver	0″83 ± 0″01	0″83 ± 0″01	0.41	0.97
gesummv	0″61 ± 0″00	0″61 ± 0″00	0.96	0.97
gramschmidt	0″86 ± 0″03	0″88 ± 0″03	-2.74	2.29
heat-3d	0″91 ± 0″01	0″90 ± 0″01	0.93	1.70
jacobi-1d	0″57 ± 0″00	0″56 ± 0″00	1.51	0.97
jacobi-2d	0″85 ± 0″03	0″87 ± 0″03	-2.49	2.02
lu	0″76 ± 0″01	0″73 ± 0″01	3.97	1.01
ludcmp	0″75 ± 0″01	0″76 ± 0″01	-1.40	1.83
mvt	0″78 ± 0″01	0″78 ± 0″01	-0.13	0.97
nussinov	0″86 ± 0″03	0″88 ± 0″03	-3.04	0.94
seidel-2d	0″89 ± 0″01	0″86 ± 0″01	3.31	0.99
symm	0″87 ± 0″01	0″88 ± 0″01	-1.33	2.14
syr2k	0″88 ± 0″01	0″90 ± 0″01	-2.25	2.13
syrk	0″88 ± 0″01	0″87 ± 0″01	0.41	1.36
trisolv	0″57 ± 0″00	0″56 ± 0″00	1.94	0.96
trmm	0″86 ± 0″02	0″90 ± 0″02	-4.32	2.32
Mean			1.96	1.46

Table 8: Single issue experiment results, averaged over 5 runs. Runs in red experienced no multifidelity behavior.

Benchmark	IPC_{true}	IPC_{mf}	Pct. Error	Speedup
2mm	1″63 ± 0″01	1″66 ± 0″01	-1.54	2.01
3mm	1″64 ± 0″01	1″74 ± 0″01	-5.90	0.99
adi	1″60 ± 0″02	1″63 ± 0″02	-1.84	1.03
atax	1″27 ± 0″01	1″31 ± 0″01	-3.38	0.99
bicg	1″22 ± 0″02	1″28 ± 0″02	-4.71	1.00
cholesky	1″44 ± 0″02	1″48 ± 0″02	-2.65	1.01
correlation	1″61 ± 0″03	1″75 ± 0″03	-8.10	2.47
covariance	1″64 ± 0″03	1″49 ± 0″03	10.15	2.77
deriche	1″49 ± 0″02	1″55 ± 0″02	-3.38	1.03
doitgen	1″63 ± 0″02	1″73 ± 0″02	-5.67	2.25
durbin	1″00 ± 0″02	1″08 ± 0″02	-6.79	0.99
fdtd-2d	1″59 ± 0″02	1″62 ± 0″02	-2.24	1.02
floyd-warshall	1″64 ± 0″03	1″63 ± 0″03	1.02	1.25
gemm	1″60 ± 0″02	1″59 ± 0″02	0.62	2.09
gemver	1″45 ± 0″01	1″51 ± 0″01	-4.14	1.01
gesummv	0″87 ± 0″02	0″94 ± 0″02	-7.60	0.98
gramschmidt	1″61 ± 0″01	1″73 ± 0″01	-6.95	2.45
heat-3d	1″72 ± 0″03	1″70 ± 0″03	0.97	1.38
jacobi-1d	0″78 ± 0″02	0″84 ± 0″02	-6.89	1.00
jacobi-2d	1″64 ± 0″04	1″74 ± 0″04	-5.98	2.15
lu	1″45 ± 0″02	1″47 ± 0″02	-1.20	1.03
ludcmp	1″42 ± 0″02	1″46 ± 0″02	-3.08	1.02
mvt	1″32 ± 0″01	1″38 ± 0″01	-4.45	0.99
nussinov	1″64 ± 0″03	1″77 ± 0″03	-7.55	0.97
seidel-2d	1″67 ± 0″01	1″74 ± 0″01	-4.10	1.26
symm	1″65 ± 0″02	1″69 ± 0″02	-2.00	2.29
syr2k	1″68 ± 0″03	1″64 ± 0″03	2.48	2.26
syrk	1″65 ± 0″03	1″75 ± 0″03	-5.85	1.02
trisolv	0″79 ± 0″02	0″85 ± 0″02	-7.19	0.98
trmm	1″67 ± 0″03	1″58 ± 0″03	6.27	2.54
Mean			4.49	1.47

Table 9: Double issue experiment results

Benchmark	IPC_{true}	IPC_{mf}	Pct. Error	Speedup
2mm	0″90 ± 0″00	0″88 ± 0″00	1.87	2.39
3mm	0″89 ± 0″00	0″92 ± 0″00	-3.33	0.97
adi	0″90 ± 0″00	0″90 ± 0″00	0.19	1.01
atax	0″68 ± 0″01	0″69 ± 0″01	-1.43	0.97
bicg	0″67 ± 0″00	0″68 ± 0″00	-0.98	0.97
cholesky	0″64 ± 0″00	0″63 ± 0″00	1.31	1.01
correlation	0″89 ± 0″01	0″91 ± 0″01	-2.36	2.37
covariance	0″89 ± 0″00	0″90 ± 0″00	-1.20	2.16
deriche	0″87 ± 0″00	0″88 ± 0″00	-0.94	1.43
doitgen	0″89 ± 0″00	0″90 ± 0″00	-1.05	2.17
durbin	0″52 ± 0″01	0″53 ± 0″01	-2.03	0.96
fdtd-2d	0″90 ± 0″00	0″90 ± 0″00	0.22	1.00
floyd-warshall	0″89 ± 0″00	0″89 ± 0″00	-0.43	2.12
gemm	0″89 ± 0″00	0″89 ± 0″00	-0.80	2.07
gemver	0″80 ± 0″00	0″78 ± 0″00	2.00	1.13
gesummv	0″47 ± 0″00	0″48 ± 0″00	-3.27	0.97
gramschmidt	0″89 ± 0″00	0″91 ± 0″00	-1.50	2.34
heat-3d	0″91 ± 0″00	0″92 ± 0″00	-0.88	1.90
jacobi-1d	0″41 ± 0″00	0″40 ± 0″00	2.09	0.96
jacobi-2d	0″89 ± 0″00	0″90 ± 0″00	-1.05	2.21
lu	0″64 ± 0″01	0″62 ± 0″01	2.40	1.03
ludcmp	0″63 ± 0″00	0″64 ± 0″00	-1.28	1.24
mvt	0″72 ± 0″00	0″71 ± 0″00	0.53	1.11
nussinov	0″89 ± 0″01	0″92 ± 0″01	-2.89	0.97
seidel-2d	0″91 ± 0″00	0″92 ± 0″00	-1.15	2.17
symm	0″90 ± 0″00	0″91 ± 0″00	-1.11	2.26
syr2k	0″90 ± 0″00	0″91 ± 0″00	-1.24	1.98
syrk	0″90 ± 0″00	0″92 ± 0″00	-1.82	2.38
trisolv	0″42 ± 0″00	0″42 ± 0″00	1.69	0.97
trmm	0″89 ± 0″00	0″91 ± 0″00	-2.04	2.44
Mean			1.50	1.59

Table 10: TimingDRAM experiment results

Block size	IPC_{true}	IPC_{mf}	Pct. Error	Speedup
1	0″88 ± 0″01	0″88 ± 0″01	-0.12	2.26
2	0″88 ± 0″00	0″88 ± 0″00	-0.25	2.13
3	0″88 ± 0″00	0″89 ± 0″00	-1.10	2.09
4	0″87 ± 0″01	0″89 ± 0″01	-1.68	2.09
5	0″88 ± 0″00	0″89 ± 0″00	-0.75	2.07
6	0″88 ± 0″00	0″89 ± 0″00	-0.96	2.02
9	0″88 ± 0″00	0″89 ± 0″00	-0.81	1.99
10	0″88 ± 0″00	0″88 ± 0″00	-0.51	1.96
12	0″88 ± 0″00	0″89 ± 0″00	-0.76	1.94
15	0″88 ± 0″00	0″88 ± 0″00	-0.01	1.95
18	0″88 ± 0″01	0″87 ± 0″01	1.18	1.88
20	0″88 ± 0″01	0″88 ± 0″01	-0.43	1.92
30	0″88 ± 0″01	0″88 ± 0″01	-0.34	1.86
36	0″87 ± 0″01	0″87 ± 0″01	1.09	1.92
45	0″88 ± 0″00	0″91 ± 0″00	-3.01	1.05
60	0″88 ± 0″00	0″89 ± 0″00	-0.20	2.16
90	0″89 ± 0″00	0″89 ± 0″00	-0.06	2.35
Mean			0.78	1.98

Table 11: Matrix multiply experiment results

Stride	IPC_{true}	IPC_{mf}	Pct. Error	Speedup
1	0″62 ± 0″00	0″61 ± 0″00	1.80	1.46
2	0″62 ± 0″00	0″62 ± 0″00	-0.42	1.00
3	0″61 ± 0″00	0″60 ± 0″00	1.02	1.38
4	0″61 ± 0″00	0″61 ± 0″00	0.74	0.99
5	0″59 ± 0″00	0″53 ± 0″00	11.90	1.46
6	0″60 ± 0″00	0″59 ± 0″00	2.21	1.51
7	0″58 ± 0″00	0″58 ± 0″00	-0.04	1.35
8	0″58 ± 0″00	0″58 ± 0″00	1.49	1.46
Mean			2.45	1.33

Table 12: Spatter experiment results

Received 8 August 2023; revised 7 September 2023